SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing

Thomas Ropars
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
thomas.ropars@epfl.ch

Tatiana V. Martsinkevich
INRIA, University of Paris Sud
Paris, France
tatiana.mar@inria.fr

Amina Guermouche
Université de Versailles Saint-Quentin en Yveline
Versailles, France
amina.guermouche@uvsq.fr

André Schiper
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
andre.schiper@epfl.ch

Franck Cappello
Argonne National Laboratory
Argonne, USA
cappello@mcs.anl.gov

ABSTRACT

The high failure rate expected for future supercomputers requires the design of new fault tolerant solutions. Most checkpointing protocols are designed to work with any message-passing application but suffer from scalability issues at extreme scale. We take a different approach: We identify a property common to many HPC applications, namely *channel-determinism*, and introduce a new partial order relation, called *always-happens-before* relation, between events of such applications. Leveraging these two concepts, we design a protocol that combines an unprecedented set of features. Our protocol called SPBC combines in a hierarchical way coordinated checkpointing and message logging. It is the first protocol that provides failure containment without logging any information reliably apart from process checkpoints, and this, without penalizing recovery performance. Experiments run with a representative set of HPC workloads demonstrate a good performance of our protocol during both, failure-free execution and recovery.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Checkpoint/restart

General Terms

Algorithms, Reliability

1. INTRODUCTION

As High Performance Computing (HPC) systems keep growing in scale, providing efficient fault tolerance mechanisms becomes a major issue. Studies on future exascale systems highlight that, considering that the expected mean time between failures (MTBF) will range from one day to a few hours, simple solutions based on coordinated checkpoints saved to a parallel file system (PFS) will not work: more time will be spent dealing with failures than doing useful computation [11, 14, 28].

A checkpointing protocol for large scale HPC systems should provide good performance in failure-free execution and in recovery while limiting the amount of resources used for fault tolerance. These goals can be conflicting. Dealing with high failure rate requires a high checkpointing frequency to limit the extent of rollbacks in time. But increasing the frequency can also impact the failure-free performance. Message logging can be used to avoid rolling back all the processes but it implies saving the message payload in the node memory [21], while the memory size per CPU available in future exascale systems is going to be smaller compared to the current situation. Coordinated checkpointing, on the other hand, does not require saving any messages in the nodes' memory but if a failure occurs all processes are required to roll back to the last checkpoint.

Hybrid protocols, combining coordinated checkpointing and message logging, have been recently proposed for fault tolerance at large scale [7, 25, 29]. A hierarchical protocol that applies coordinated checkpointing inside clusters of processes and message logging between clusters, can limit the impact of a failure to one cluster while logging only a subset of messages. Existing hierarchical solutions still have some limitations related to message logging: To be able to replay messages in a valid order after a failure, they need to log all non-deterministic events related to message delivery occurring during execution. However, event logging, even when implemented in a distributed way, implies a use of additional resources [29] and can impair failure-free performance [31]. The authors of [29] propose not to log events reliably to improve event logging performance, but they do not ensure failure containment if some events are lost. In this paper we propose a hierarchical protocol that does not log any events during the failure-free execution, but still ensures failure containment.

The protocols described in [7, 25, 29] work with all message-passing applications. In this paper, we take a different approach: We identify properties common to our tar-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13, November 17 - 21 2013, , USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503271.

get applications, namely SPMD MPI HPC applications, and we leverage these properties to design a fault tolerant solution that can be more efficient than existing protocols at large scale. According to our observation, scientific SPMD applications are composed of a well-defined set of computation and communication patterns that can be interleaved or overlapped. After a failure rolled back processes have to replay some communication patterns. It is then necessary to ensure that logged messages are delivered in the communication pattern or iteration of a pattern they belong to. Note that in an MPI application incorrect message delivery can only happen if the `MPI_ANY_SOURCE` wildcard is used.

To formalize our observation, we introduce two new concepts. First, we introduce a new property that we found to be common to many MPI HPC applications, called *channel-determinism*. It states that for a given set of input parameters the sequence of messages sent on communication channels is the same in any valid execution. This property extends the previously defined *send-determinism* property [9] and states that execution path of most scientific applications is not impacted by the relative order of message delivery. Thus we conclude that event logging is not required since we do not have to ensure the total order of message delivery during recovery. The second concept is *always-happens-before* relation. It defines the causal dependency between messages that are inherent to the program and allows us to formalize the previously mentioned notions of *patterns* and *iterations*.

While existing checkpointing protocols consider the *happened-before* relation [23] that orders events of an execution, we show how the *always-happens-before* relation can be used to design a simple protocol for channel-deterministic applications. Channel-determinism allows avoiding logging information about the reception order of messages during failure-free execution, and the information about *always-happens-before* relations is used for deciding on which message to receive next during recovery when `MPI_ANY_SOURCE` is used.

The contribution of our work can be summarized as follows. We introduce *channel-determinism* and *always-happens-before* relation in Section 3. We present our protocol called Scalable Pattern-Based Checkpointing (SPBC) in Section 4. We describe our implementation of SPBC in MPICH in Section 5. This section also introduces our programming interface that can be used to explicitly describe *always-happens-before* relations in the code where an incorrect message delivery may occur during recovery. Finally, we evaluate the performance of SPBC in failure-free execution and in recovery on a representative set of HPC benchmarks and applications in Section 6. Our evaluation shows that 1) our solution requires only few (if any) modifications of the applications; 2) SPBC is efficient both in failure-free execution and in recovery.

2. STATE OF THE ART

In this section, we position our work with respect to related work. We present the standard fault-tolerant solution based on coordinated checkpointing, and then discuss the use of message logging for providing failure containment.

2.1 Coordinated Checkpointing in HPC Systems

Coordinated checkpointing is commonly used in HPC systems for dealing with crash failures, because it is easy to implement and it does not require any data to be saved apart from process checkpoints. However, it has two main disadvantages that can impair its scalability [14, 28], namely, the contention it may introduce in the PFS and the lack of failure containment. Considering that coordinated checkpointing can cause more than 50% overhead in performance, replication has been considered as a possible alternative [16]. However, with recent improvement in the checkpointing techniques, replication is often considered to be too costly for dealing with crash failures [29]. But it still can be of interest for handling silent errors [17].

The relatively low bandwidth of the PFS means that saving or retrieving a checkpoint of the whole application state can be very slow (in the order of tens of minutes [27]) and will impact both failure-free and recovery performance. Multi-level checkpointing solutions based on local storage and erasure codes have been proposed [3, 27] to deal with this issue. They allow high checkpoint frequency while achieving good failure-free performance. Note that multi-level checkpointing can be efficiently combined with hybrid checkpointing protocols [4].

Another disadvantage of coordinated checkpointing is that when a failure occurs, all processes roll back to the last checkpoint. Causal dependencies between processes can be taken into account to roll back only the processes that causally depends on the failed one [22]. However, in MPI HPC applications, all processes become causally dependent on each other very fast [18]. Rolling back all the processes is costly because it may cause an IO burst when retrieving the last checkpoint. Additionally, re-executing the lost computations after a rollback is a big waste of resources and, consequently, of energy.

Restarting from the last coordinated checkpoint, *i.e.*, from a consistent global state, allows to deal with non-deterministic applications [15]. But failure containment, *i.e.*, limiting the consequences of a failure to a subset of processes, is a key to fault tolerance at extreme scale [11, 10]. Providing failure containment with a checkpointing protocol requires introducing message logging.

2.2 Message Logging

Message logging can be used to limit the extent of rollbacks in space [15]. Messages are logged during failure-free execution and replayed in the same order after a failure. Message delivery events need to be saved reliably while the payload can be saved in the sender's memory [21]. Message logging protocols (pessimistic or causal) provide perfect failure containment because only the failed process has to roll back. But this comes at the cost of a high overhead in failure-free execution [31]. On the other hand, optimistic message logging protocols, where events are logged asynchronously, perform better but failure containment is ensured only if no event is lost after a failure.

Even when optimistic message logging is used, the cost of logging events reliably can be unacceptably high at scale [31]. It has been proposed to limit the number of events to log by taking into account the semantics of MPI calls and to distinguish events by logging only the ones that correspond to a non-deterministic behavior of the MPI library (*e.g.*, use of `MPI_ANY_SOURCE` or `MPI_Waitany`) [5]. However, there is no evidence that such a solution would be efficient enough at a very large scale. To improve the performance of optimistic event logging, it has been recently proposed to simply log

the events in the memory of a node and to combine message logging with coordinated checkpointing to avoid the domino effect in case many events are lost due to a failure [29]. It should be mentioned that actually most recent evaluations of message logging protocols implement logging in the memory of dedicated nodes [6, 8].

In a message logging protocol, message payload is saved in the node memory. For some applications, logs can grow very fast leading to a huge memory use. An alternative is to use a hybrid protocol that combines coordinated checkpointing and message logging protocols hierarchically: processes are partitioned into groups, or clusters, and coordinated checkpointing is used within a cluster, while message logging is used for inter-cluster messages [7, 19, 25]. Such a hybrid approach can dramatically reduce the amount of data to be logged [30].

A hybrid protocol provides failure containment by ensuring that if a failure occurs, only the processes in the failed cluster have to roll back. However, it still requires all non-deterministic events, even those related to messages exchanged inside a cluster, to be logged during failure-free execution to ensure that the exact same execution can be reproduced after a failure [7]. Thus, the performance of these protocols can also be impaired by the overhead of event logging. To deal with this issue, [19] proposes a hybrid protocol that leverages the *send-determinism* common to many MPI HPC applications [9]. In a send-deterministic application, a process sends the same sequence of messages in any valid execution for a given set of input parameters (see Definition 1). This protocol does not require to log any events during failure-free execution and can leverage the send-determinism property to correctly re-order most messages during recovery. However, in cases where the process to receive from is not specified in the call (use of `MPI_ANY_SOURCE`), processes need to synchronize during the replay of logged messages to ensure that a message is re-sent only when all messages it causally depends on have been replayed. Our experiments show that this synchronization can slow down the recovery after a failure (see Section 6.5).

In this paper, we leverage two newly defined concepts for MPI HPC applications, namely *channel-determinism* and *always-happens-before* relation, and propose a hybrid protocol that does not log any delivery events during failure-free execution and does not require synchronization between replaying processes during recovery. In contrast to [29], our protocol always guarantees that the consequence of a failure is limited to a subset of processes. However, our solution is limited to *channel-deterministic* applications while [29] works with all applications. In the next section we define *channel-determinism* and the *always-happens-before* relation.

3. PROPERTIES OF MPI HPC APPLICATIONS

We start by presenting our message-passing application model. Then we introduce the concepts related to MPI. Finally, we present the *channel-deterministic* property and the *always-happens-before* relation.

3.1 Model

To model a message passing parallel algorithm, we consider a set $P = \{p_1, p_2, ..., p_n\}$ of n processes, and a set C of channels connecting any ordered pair of processes. Channels are assumed to be FIFO and reliable but no assumption is made about the system synchrony. The channel from processes p_i to p_j is denoted c_{ij}.

Suppose an execution $exec(A)$ of an algorithm A whose initial state is $\Sigma^0 = \{\sigma_1^0, \sigma_2^0, ..., \sigma_n^0\}$, where σ_i^0 is the initial state of process p_i. An execution is driven by a set $S_{exec(A)}$ of events e_i^k, where e_i^k is the k^{th} event in process p_i. The state of process p_i after the occurrence of e_i^k is σ_i^k. The events in $S_{exec(A)}$ are partially ordered by Lamport's *happened-before* relation [23], denoted $\rightarrow$.

Starting from initial state Σ^0 an algorithm may go along different execution paths. We define $\mathscr{E}_A$ as a set of execution paths that A can follow when no failure occurs: $\mathscr{E}_A$ is the set of valid executions of A. The set $\mathscr{S}_A$ includes the partially-ordered sets of events S_E corresponding to the executions E in $\mathscr{E}_A$. The sub-sequence of S_E consisting of events on process p_i is denoted $S_E|p_i$. Similarly, the sub-sequence of S_E consisting of events on channel c is denoted $S_E|c$

Note that in this paper we consider crash failures and we assume that multiple concurrent failures can occur.

3.2 MPI Applications

For a detailed description of MPI, we refer the reader to the MPI specification [26]. Here we provide a short description of the main concepts related to point-to-point communication. MPI also provides primitives for collective operations but, unless hardware-specific information is provided, we assume that collective operations are implemented on top of point-to-point communication.

An MPI message is defined by a payload and a message envelope (the metadata). The metadata contains message identification information: the source identifier (src), the destination identifier (dst), a tag (tag) and a communicator ($comm$). The communicator specifies the communication context of the message. Any MPI communication operation is associated with a communicator. Hence, to adapt the general model presented in Section 3.1, we consider that a channel is also defined in the context of a communicator: There can be multiple channels between two processes, one for each communicator they belong to.

Figure 1 shows the events we associate with a point-to-point MPI communication. It describes a scenario where a process p_1 sends a message m to a process p_2. It shows the general case when a process uses `MPI_Isend` (resp. `MPI_Irecv`) to post a *send* (resp. *recv*) request and then, `MPI_Wait` to wait for the request to complete.

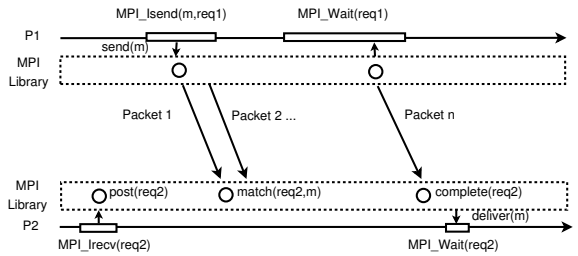

Figure 1: MPI point-to-point communication

At the MPI library level, we associate a `post(req)` and a `complete(req)` event with the time a reception request is posted to the library and completed at the library level respectively. We also introduce two events at the application

level: an event `send(m)` corresponds to the time when the application process calls the MPI function to initiate the sending of message m; an event `deliver(m)` corresponds to the time when a received message m is available to the application process.

We define a last event at the MPI level, `match(req, m)`, to represent the matching between an incoming message m and a posted reception request req. In a reception request the source, tag and communicator of the message to receive can be specified. A request is matched with the first received message (based on the first packet of the message) whose metadata matches the request. The FIFO property of MPI channels guarantees that if two messages are sent through the same channel and they both can be matched with the same request, the first sent message will be matched first[1].

There are two main sources of non-determinism in MPI communication. First, instead of defining the source in a reception request, one can use the wildcard `MPI_ANY_SOURCE` to receive the next message from any sender. Second, among the functions that check the completion of requests, some of them show non-deterministic behavior that depends on the speed of message arrival (*e.g.*, `MPI_Test`, `MPI_Waitany`).

3.3 Identifying Events

We want to be able to compare events related to MPI communication in different executions of an application. To do so, we introduce a per-channel sequence number. This *seqnum* is implicit in the standard, but it exists in all MPI libraries to ensure the FIFO property. Thus, a message m is uniquely identified by a payload and a tuple $\{src, dst, comm, seqnum\}$ of metadata. We say that message m_1 of execution E_1 and message m_2 of execution E_2 of algorithm A are the same if their payload and metadata are the same. We compare the events associated with sending and receiving of messages in the same way: $send(m_1)$ and $send(m_2)$ are the same if m_1 and m_2 are the same. To be able to compare reception requests from different executions, we also associate each request with a sequence number given at the time the request is posted. Thus, a reception request is uniquely identified by a tuple $\{src, dst, comm, seqnum\}$.

3.4 Channel-Determinism

To design an efficient checkpointing protocol for HPC applications, we propose to design a protocol that can work for most HPC applications, instead of designing a protocol that can work for all distributed applications. For this reason we consider only SPMD applications as defined in [24] and master-worker applications are excluded.

As it has been shown in [9], SPMD MPI applications are mostly deterministic even if the MPI interface allows some non-deterministic behavior. More precisely, [9] defined the send-deterministic property:

Definition 1 (Send-deterministic algorithm). *An algorithm A is send-deterministic if, given an initial state Σ^0, for each $p \in P$ and $\forall S, S' \in \mathscr{S}_A$, $S|p$ and $S'|p$ contain the same subsequence of* send *events.*

Essentially it means that the relative order of message receptions has no impact on the order and on the content of the messages sent by a process. This definition enforces a total order of all the messages sent by a process. We introduce channel-determinism as a new property that enforces the total order only per channel:

Definition 2 (Channel-deterministic algorithm). *An algorithm A is channel-deterministic if, considering an initial state Σ^0, for each channel $c \in C$ and $\forall S, S' \in \mathscr{S}_A$, $S|c$ and $S'|c$ contain the same sub-sequence of* send *events.*

Obviously, all send-deterministic applications are channel-deterministic. But channel-determinism encompasses more communication patterns. For instance, the communication pattern illustrated in Figure 4, that can be found in BoomerAMG [20], is channel-deterministic but not send-deterministic (see Section 5.1).

Send-determinism was used in [18] and [19] to avoid logging order of message delivery in a message logging protocol. However, note that these protocols would not work with *channel-determinism* because they rely on the per-process total order of send events to infer *happened-before* relations between messages during recovery. If two send events should occur in a different order after a failure, wrong *happened-before* relations will be inferred, potentially leading to a deadlock. Our protocol described in Section 4 is not based on the *happened-before* relation.

3.5 The Always-Happens-Before Relation

Channel-determinism implies that the set of messages exchanged in any failure-free execution of an algorithm is the same. Assuming that a process does not cancel requests and that all posted requests are eventually completed, we can also assume that the set of reception requests used to receive these messages is also always the same. Thus, given an initial state Σ^0, in any valid execution of a channel-deterministic algorithm the set of *send*, *post* and *deliver* events is the same. We call it the set of *communication events*. We also include *match* events in this set. Such events may differ from one execution to another because a reception request is not necessarily always matched with the same message. However, if we denote $match(req, _)$ the event associated with the matching of request req, such an event can be found in all valid executions of an algorithm.

We define the *always-happens-before* relation as a partial order relation on the set of communication events of a channel-deterministic algorithm. This relation is defined over the set of valid executions of the algorithm:

Definition 3 (Always-happens-before relation). *Consider an algorithm A, all executions in $\mathscr{E}_A$, and assume that e_1, e_2 are two communication events existing in all executions in $\mathscr{E}_A$. If $e_1 \rightarrow e_2$ in all executions, then we say that e_1* always-happens-before *e_2, denoted $e_1 \overset{A}{\rightarrow} e_2$.*

Note that the *always-happens-before* relation and the *happened-before* relation are not equivalent. The *always-happens-before* relation is a property of the algorithm A, whereas the *happened-before* relation applies to an execution of A. The *always-happens-before* relation aims at expressing the causal relations between messages that a programmer describes in the design of her algorithm.

4. THE SPBC PROTOCOL

This section describes the SPBC protocol. We seek to obtain a hierarchical protocol that does not need to log reception events to provide failure containment, and that also

[1]Note that it does not guarantee that the message reception is completed first.

does not require the processes to synchronize during recovery to correctly re-order logged messages in contrast to [19]. The basic SPBC protocol is described in Algorithm 1. By default this protocol may not work with all channel-deterministic algorithms, especially if it includes `MPI_ANY_SOURCE` wildcards, as we explain in Section 4.2. Such an algorithm A has *always-happens-before* relations in failure-free execution that the SPBC protocol may not be able to guarantee after a failure, leading to an invalid execution. In Section 4.3, we explain how A should be modified into an algorithm A' to ensure that SPBC is always able to ensure a valid execution with respect to the *always-happens-before* relation despite crashes.

4.1 Description of the Protocol

SPBC is a hierarchical protocol where processes are partitioned into clusters and coordinated checkpointing is used inside each cluster (line 14) while inter-cluster messages are logged in the memory of their sender (line 6). When a failure occurs, all processes belonging to the cluster where the process crashed roll back to their last checkpoint (line 18). Processes in other clusters replay missing messages from the logs in the same order as they were sent during failure-free execution (line 24). The sequence number of the last received message ($c_{ji}.LR$) is stored for each incoming channel to know which messages need to be replayed from the logs after a failure (line 23). The sequence number of the last sent message ($c_{ij}.LS$) is stored for each outgoing channel to know which messages a process that rolls back has to resend during recovery (line 7): If the destination has already received a message in the current state, it is not needed to send it again.

It can easily be shown that such a protocol cannot lose messages. Coordinated checkpointing ensures that the state of intra-cluster channels remains consistent despite failures. On inter-cluster channels, messages are either available in the logs or will be regenerated during recovery if the sender rolled back before sending the message. However, unlike hybrid protocols [7, 25] that save the order of message delivery during failure-free execution, the protocol described in Algorithm 1 could lead to an invalid execution after a failure if non-deterministic MPI calls are used by the application because it does not know how to correctly order logged messages coming from different channels.

Algorithm 1 SPBC protocol for process p_i

Local Variables:
1: $cluster_i$ /* ID of the cluster of process i */
2: $Logs_i \leftarrow \emptyset$

3: **Sending message** msg **on channel** c_{ij}
4: $c_{ij}.seqnum \leftarrow c_{ij}.seqnum + 1$
5: **if** $cluster_j \neq cluster_i$ **then**
6: $Logs_i \leftarrow Logs_i \cup (c_{ij}, c_{ij}.seqnum, msg)$
/* Avoiding sending unnecessary messages during recovery */
7: **if** $c_{ij}.seqnum > c_{ij}.LS$ **then**
8: Send $(msg, c_{ij}.seqnum)$ on channel c_{ij}
9: $c_{ij}.LS \leftarrow c_{ij}.seqnum$

10: **Upon** $(msg, seqnum)$ **on channel** c_{ji}
11: $c_{ji}.LR \leftarrow seqnum$
12: Deliver msg to the application

13: **Upon checkpoint**
14: Execute Coordinate Protocol inside $cluster_i$
15: Save $(State_i, Logs_i)$ on stable storage

16: **Upon failure of process** P_j
17: **if** $cluster_i = cluster_j$ **then**
18: Restart from last $(State_i, Logs_i)$ on stable storage
19: **for all** outgoing channels c_{ij} **such that** $cluster_j \neq cluster_i$ **do**
20: Send(`Rollback`, $c_{ij}.LR$) on c_{ij}

21: **Upon** (`Rollback`, $seqnum_{rb}$) **on channel** c_{ji}
22: Send(`lastMessage`, $c_{ji}.seqnum$) on c_{ij}
23: **for all** $(c_{ij}, seqnum, msg) \in Log_i$ **such that** $seqnum > seqnum_{rb}$ **do**
24: Send $(msg, seqnum)$ on c_{ij} /* messages are sent in the order of their sequence number */

25: **Upon** (`lastMessage`, $seqnum$) **on channel** c_{ji}
26: $c_{ji}.LS \leftarrow seqnum$

4.2 Difficult Cases for Recovery

We identified two cases that could create problems during recovery. The first case is related to the use of `MPI_ANY_SOURCE`. The second case is related to the use of non-deterministic completion functions. To illustrate these cases, we consider a scenario with three processes (see Figure 2). Processes p_0 and p_1 are in one cluster while process p_2 is in another cluster. There is an *always-happens-before* relation between $deliver(m_0)$ and $deliver(m_2)$: During failure-free execution, m_1 cannot be sent before m_0 is received and m_2 cannot be sent before m_1 is received. We describe two cases where processes p_0 and p_1 roll back after a failure, and where during recovery $deliver(m_2)$ could occur before $deliver(m_0)$ if m_2 is replayed from the logs.

4.2.1 Case 1: Faulty matching between requests and messages

In Figure 2, process p_1 uses anonymous reception requests to receive m_0 and m_2. During recovery, message m_0 is resent by p_0 during its re-execution whereas message m_2 is replayed by p_2 from its logs. Thus, p_2 does not have to wait for m_0 to be replayed to send m_2. If m_2 is received before m_0 at the MPI level, it is delivered to the process p_1 first, leading to an invalid execution. Note that in this scenario, if p_1 did not use `MPI_ANY_SOURCE` to receive messages but explicitly tried to receive first from p_0 and then from p_2, there would be no problem during recovery.

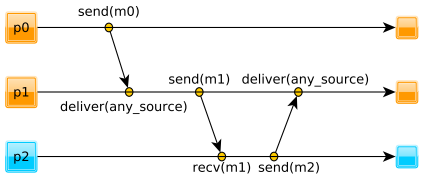

Figure 2: Scenario with `MPI_ANY_SOURCE`

4.2.2 Case 2: Faulty request completion order

Even if anonymous reception requests are not used, some scenarios can lead to an invalid execution. These scenarios involve non-deterministic completion functions. In Figure 3 we assume that process p_1 posts two named reception requests to receive messages m_0 and m_2 respectively. Then p_1 checks the completion of the two requests at the same time using `MPI_Waitany`. During failure-free execution, since m_2 cannot be sent before m_0 is received, m_0 will always be delivered first. However, during recovery, since m_2 is replayed

from the logs, it may be delivered first, leading again to a non-valid execution.

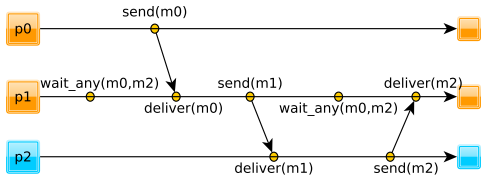

Figure 3: Scenario with `MPI_Waitany`

In this case, the problem comes from the fact that the process is trying to concurrently complete the reception requests for messages m_0 and m_2, but $deliver(m_0) \overset{A}{\rightarrow} deliver(m_2)$. Obviously, trying to complete the two requests concurrently is useless since m_2 cannot be received before m_0 in a correct execution. It is important to mention that this situation is different from a case where `MPI_Waitany` is used to complete the requests from a set of *real* concurrent reception requests as soon as possible, *i.e.*, to receive from a set of messages that can arrive in any order during a correct execution. Here the reception of m_0 and m_2 are not concurrent in the sense that $deliver(m_0) \overset{A}{\rightarrow} deliver(m_2)$. We think that in this specific case, `MPI_Waitany` can always be replaced by `MPI_Wait` without harming performance, and so, we choose not to handle this case.

4.3 Matching Based on IDs

To avoid faulty matching between messages and requests during recovery with SPBC, we propose to add an extra identifier to every message and to every reception request (Algorithm A'). We also specify the conditions that such identifiers should implement to ensure that the execution of A' with SPBC will always lead to an execution in $\mathscr{E}_A$ despite failures.

The scenario described in Section 4.2.1 shows that a request can be matched with only some messages in a valid execution of an algorithm. For each request in the application, we can define a *matching set*:

Definition 4 (RECEPTION REQUEST MATCHING SET). *A message m is in the matching set M_{req} of a reception request req if, given an initial state Σ^0, $\exists E \in \mathscr{E}_A$ that includes event $match(req, m)$.*

We say that message m and request req are mismatched if, during recovery, m is matched with req although m is not in the matching set of req. To avoid the mismatch after a failure, we propose to add an extra identifier to each message ($id(m)$) and to each request ($id(req)$) in the application, and to allow the matching between a request and a message only if they have the same identifier. To ensure that mismatches cannot occur during recovery, it is enough to implement the two following conditions:

1. $\forall req$, $\forall m \in M_{req}$, $id(m) = id(req)$.
2. If m and req can be mismatched after a failure, then $id(m) \neq id(req)$.

The second condition could be replaced by a stronger one. Namely, if $m \notin M_{req}$, then $id(m) \neq id(req)$. However, this condition would require more efforts to be implemented. Moreover, this condition is obviously not necessary: if no `MPI_ANY_SOURCE` wildcard is used, requests and messages cannot be mismatched even during recovery. To refine the second condition, we formulate the following theorem:

Theorem 1. *Using SPBC, a message m and a request req can be mismatched during recovery if and only if req is an anonymous reception request and $match(req, _) \overset{A}{\rightarrow} match(_, m)$.*

PROOF. It should be mentioned that the mismatch we consider is the *first* occurring in the execution. Of course, once one mismatch has occurred, other mismatches can occur and involve even named reception requests.

Since applications are channel-deterministic and we do not consider the problem identified in Section 4.2.2, a mismatch can only involve an anonymous reception request. Indeed, if no anonymous request is used, the matching between messages and requests is done per channel. Since logged messages are re-sent on the channel in the same order as during the failure-free execution, an execution where messages are replayed from the logs is indistinguishable from a failure-free execution when channels are considered separately.

Then, to show that a mismatch can occur between a message m and an anonymous request req only if $match(req, _) \overset{A}{\rightarrow} match(_, m)$, we show that it cannot occur in any other cases. The first case is $match(_, m) \overset{A}{\rightarrow} match(req, _)$. The second case is when there is no *always-happens-before* relation between req and m, that is $match(_, m) \overset{A}{\not\rightarrow} match(req, _)$ and $match(req, _) \overset{A}{\not\rightarrow} match(_, m)$

We first assume an anonymous request req and a message $m \notin M_{req}$ such that $match(_, m) \overset{A}{\rightarrow} match(req, _)$. By definition, in any valid execution $match(_, m) \rightarrow match(req, _)$. Since $match(req, _)$ does not depend on any mismatch, the execution is valid until $match(req, _)$ occurs. It implies that $match(_, m)$ has already occurred when $match(req, _)$ occurs. Since a message can be matched with only one request, m cannot be mismatched with request req.

Then we consider the case where (a) $match(_, m) \overset{A}{\not\rightarrow} match(req, _)$ and (b) $match(req, _) \overset{A}{\not\rightarrow} match(_, m)$, and we show that $m \in M_{req}$. Indeed, we can deduce from (b) that $\exists E_1 \in \mathscr{E}_A$ where $match(_, m) \rightarrow match(req, _)$. Thus, m can be received at the MPI level before req is posted. But we deduce from (a) that m is not always matched when $match(req, _)$ occurs. Thus, since req is an anonymous reception request, $\exists E_3 \in \mathscr{E}$ where req is matched with m. □

Based on Theorem 1, the two conditions for identifiers in algorithm A' can be rewritten as follows:

1. $\forall req$, $\forall m \in M_{req}$, $id(m) = id(req)$.
2. If req is an anonymous reception, then $\forall m$ such that $match(req, _) \overset{A}{\rightarrow} match(_, m)$, $id(m) \neq id(req)$.

Since these conditions are expressed using the *always-happens-before* relation, it means that they are based on the properties of algorithm A. Solutions to automatically detect *always-happens-before* relations in an MPI program is out of the scope of this paper. Therefore we do not propose an automatic transformation from algorithm A to algorithm A'. Instead, in the next section we propose an API to allow the programmer to manually do the transformation.

5. IMPLEMENTATION

This section describes our API to transform algorithm A into A' as discussed in Section 4.3. It also discusses the main points related to the implementation of SPBC in MPICH.

5.1 An API to Transform A into A'

There is a link between Theorem 1 and the way a programmer ensures the correct matching of messages and anonymous reception requests during a failure-free execution. If one logical part of the application involving communication (a communication pattern) has anonymous reception requests, the programmer has to make sure that these requests cannot be mismatched with messages from other patterns or from other iterations of the same pattern. To avoid mismatching with other patterns, one solution is to use tags as it is done in the example of Figure 4. But to avoid mismatches between iterations of a pattern that includes anonymous reception requests, the only solution is to ensure that no process can start iteration $n+1$ before all processes finished iteration n^2. This requires building an *always-happens-before* relation between messages from different iterations, *e.g.* using a barrier or another collective operation. We propose an API to make these relations explicit.

The API allows defining *communication patterns* and *iterations* inside a *pattern*. A communication pattern is defined by *pattern_id* and each iteration inside a pattern is identified by *iteration_id*. The identifier attached to each message and request is a tuple (*pattern_id*, *iteration_id*) that corresponds to the active pattern when the communication function is called. All communications that are not inside a programmer-defined pattern are associated with a *default* communication pattern. The identifiers are then used during matching between messages and requests to avoid mismatches: A message and a reception request can be matched only if they have the same identifier.

The API has three primitives. It is important to mention that these primitives do not involve any communication with other processes:

- `pattern_id DECLARE_PATTERN(void)`: generates a new *pattern_id*.
- `BEGIN_ITERATION(pattern_id)`: The pattern *pattern_id* becomes the active pattern. Its *iteration_id* is incremented by one.
- `END_ITERATION(pattern_id)`: The default communication pattern is restored.

Figure 4 illustrates how to use the API. This code is a simplified version of a pattern found in the application BoomerAMG [20]. This example is interesting because it is channel-deterministic but not send-deterministic. This code implements a data-dependent communication algorithm: Each process knows which processes it should contact based on its local data (first for-loop) but it knows neither which rank it is going to receive from nor how many ranks it is supposed to receive from [2]. Therefore, to receive messages from their neighbors the function `MPI_Iprobe` with `MPI_ANY_SOURCE` is used. A special tag *tag*1 is associated with these messages to avoid mixing them with other messages. When a process receives a message, it immediately sends a response with tag *tag*2 to the sender, which makes the code only channel-deterministic. Finally, when a process has received all the responses, it starts a termination algorithm (not described here for the sake of simplicity) to ensure that all processes received all messages.

[2]This is not a requirement added by SPBC. It is the only way to get a correct MPI code when a communication pattern includes anonymous reception requests.

```
pattern_id id=DECLARE_PATTERN();
BEGIN_ITERATION(id);
for (i = 0; i < contacts; i++){
    MPI_Irecv(buf2,.., i, tag2, rreq[i]);
    MPI_Isend(msg1,.., i, tag1,..);
}
while(!terminate){
    MPI_Iprobe(MPI_ANY_SOURCE, tag1,
               .., &probe_flag, status);
    if(probe_flag){
        proc=status.MPI_SOURCE;
        MPI_Recv(buf1,.., proc, tag1,..);
        MPI_Send(msg2,.., proc, tag2,..);
    }
    MPI_Testall(contacts, rreq, &flag);
    if(flag){/*code to detect
               global termination*/}

}
END_ITERATION(id);
```

Figure 4: Modified code including `MPI_ANY_SOURCE`

Executing several iterations of the unmodified version of this pattern in an execution without failures cannot generate mismatches between messages and requests from different iterations because the termination algorithm ensures that an iteration is finished on all processes before the next one starts. However, after a failure, if some processes should replay messages from logs, messages from iteration $n+1$ could be send before all recovering processes finish replaying iteration n, creating a risk of mismatch. Modifying the code as in Figure 4 is enough to ensure that a process that recovers from a failure would never match a message from iteration $n+1$ with a request from iteration n.

Note that being able to declare multiple patterns and to count iterations separately for each pattern is required to be able to deal with cases where only a subset of the processes is involved in some communication, *e.g.* in the context of a *sub-communicator*.

Note on the Use of the API.

Based on the example of Figure 4, one may think that using the API associated with SPBC is very restrictive with respect to the programming model and can only accommodate with *synchronous* communication patterns. It is not the case. First, recall that any application that does not include anonymous reception requests can be run as it with SPBC. In this case, it can include all kinds of asynchronous communication. Second, when an application includes anonymous reception requests, the programmer has to build *always-happens-before* relations in her code, independently from SPBC, to ensure that messages from a communication pattern including anonymous requests cannot get mixed up with other messages. It does not prevent from using asynchronous communication as long as the re-

quired *always-happens-before* relations exist. The only question that is hard to answer is whether it is easy to identify the patterns including anonymous reception requests in codes, and to mark them out using our API. Based on our experience, and considering the six applications studied in Section 6, it is usually the case.

5.2 Integration in MPICH

We implemented SPBC in MPICH-3.0.2. In the following, we describe the implementation of the matching between messages and requests based on *identifiers* and the way logged messages are replayed during recovery.

5.2.1 Matching Messages and Requests

To implement matching based on extra identifiers we defined, we include in the header of each message the tuple $(pattern_id, iteration_id)$ of the active pattern at the time the send function is called. The same thing is done for each reception request. We modified the matching function of MPICH to compare the *pattern_id* and the *iteration_id* of the message and the request, additionally to comparing the source and tag. When an inter-cluster message is logged in the memory of its sender, the corresponding identifier is also logged and replayed during recovery.

5.2.2 Replaying messages

When a failure occurs, the processes in the cluster where it happened roll back to the last checkpoint and others prepare to start replaying messages from the logs. They could immediately send all their messages to the processes that are recovering, but this would overload the recovering processes and slow down their execution. On the other hand, we do not want a recovering process to unnecessarily wait for a message that is available in the memory of its sender. We implement the following simple *flow-control* mechanism to avoid this problem.

To avoid deadlocks during recovery, we need to log in the memory of each process the total order in which the *send* requests are posted, and the order in which they are completed. Indeed, trying to complete two *send* requests that correspond to large messages (implying a *rendezvous* protocol) in an order that is different from the failure-free execution can lead to a deadlock if the dependencies inside the code imply that one cannot be completed before the other. Note also that send requests are not always completed in the order they are posted, which is why we need to log both pieces of information individually. Finally, to ensure that recovering processes will never be waiting for small messages, we allow the replaying processes to *pre-post* a set of send request before trying to complete some of them and potentially getting delayed by a *rendezvous* protocol. We observed that allowing up to 50 pre-posted messages per process was providing good performance. We used this value in the experiments presented in Section 6.

6. EVALUATION

In this section, we present a detailed performance evaluation of SPBC. Using a representative set of HPC applications and mini-applications from the NERSC8 benchmark suite[3], we evaluate its performance in failure-free execution, its memory footprint as well as its performance in recovery. We also present a comparison with the performance of HydEE [19] in recovery. Due to the current limitations of HydEE, this comparison is done using some of the NAS benchmarks.

[3]http://www.nersc.gov/systems/trinity-nersc-8-rfp/draft-nersc-8-trinity-benchmarks

6.1 Experimental Setup

Experiments are run on a 64-node cluster. Each node has two 2.5 GHz Intel Xeon CPUs (4 cores per CPU), with 16GB of memory per node. Nodes communicate over InfiniBand 20G. Operating system is Linux (kernel 3.0.0-2). Experiments are run with MPICH-3.0.2 using IPoIB.

We run a representative set of mini-applications (MiniFE, MiniGhost, Boomer-AMG, GTC, MILC) and applications (CM1). They span a variety of algorithms used in different scientific fields: MiniFE is a representative of finite element solver applications; MiniGhost implements finite difference stencil with ghost cells boundary exchange; Boomer-AMG (simply called AMG in the following) is a parallel algebraic multigrid linear solver used for problems on large unstructured grids; GTC represents a 3D Particle-in-cell application that simulates evolution of a system of gyrokinetic particles; MILC performs simulation of SU(3) lattice gauge theory used in quantum chromodynamics. Finally, CM1 is a 3D nonhydrostatic atmospheric model used for modeling atmospheric phenomena. We used the following problem sizes: 800x800x800 for MiniFE and MiniGhost, 512x100x100x100 for AMG, 1280x640x200 for CM1, and 8x8x8x8 sites per MPI tasks for MILC. GTC was run with $micell = 800$ and $npartdom = 8$. For all applications, the memory footprint per MPI process ranges between 200 and 700 MB.

Out of six applications, four (MILC, MiniFE, AMG, GTC) use anonymous receptions. We modified the code of these four applications by adding our API functions as explained above. In MILC, MiniFE and GTC, only one communication pattern was modified. In AMG, already discussed in Section 5, three patterns include `MPI_ANY_SOURCE`. For each pattern it was enough to enclose the function that contains it between a `BEGIN_ITERATION` and an `END_ITERATION` call.

Execution time presented in this section is the mean time over five executions of a test. The performance of SPBC, either failure-free or in recovery, is always compared to the native performance of the application running with the unmodified version of MPICH-3.0.2. The results are normalized to the native performance for the reference. Note that none of our experiments include checkpointing. The performance of checkpointing has been extensively studied in other papers [3, 27]. We simply recall that in SPBC the coordinated checkpoints of the different clusters can be taken independently. The goal of our evaluation is to assess the impact of message logging on performance.

To obtain the clustering configuration used in the evaluation, we ran each application for a few iterations and collected its communication statistics data. Then, we use the clustering tool presented in [30] to generate clustering configuration that minimizes the amount of data to log. All MPI processes executing on the same physical node are included in the same cluster. Indeed, providing failure containment inside a node would be useless since a node failure will result in the crash of all processes executing on that node.

6.2 Message Log Size

We start by evaluating how the size of the logs grows de-

Nb of	AMG		CM1		GTC		MILC		MiniFE		MiniGhost	
Clusters	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
2	0.1	0.4	0.1	0.8	0.1	0.9	0.1	0.1	0.1	0.1	0.3	1.1
4	0.2	0.7	0.1	0.7	0.1	0.9	0.1	0.1	0.1	0.2	0.5	2.1
8	0.4	0.7	0.2	1.5	0.2	0.9	0.2	0.2	0.1	0.3	1.1	2.1
16	0.5	0.7	0.4	1.5	0.4	0.9	0.2	0.3	0.1	0.3	1.6	2.1
64	1.2	1.4	1.5	2.2	1.7	1.7	0.4	0.4	0.2	0.3	3.7	4.2
512	1.7	2	2.8	2.9	1.7	1.8	0.6	0.6	0.5	0.6	5.5	6.3

Table 1: Logs growth rate per process in MB/s according to the number of clusters

pending on the number of clusters that are used with SBPC. As we will see in Section 6.4, this information is important to understand the performance of SPBC in recovery. We recall that logs are saved as part of the process checkpoints, and that, the associated memory can be freed afterwards.

Each application was run with 512 processes. Table 1 presents the speed with which the message logs grow per MPI process in MB/s for each application and for each clustering configuration. The table includes the average (Avg) and maximum (Max) growth rate observed for processes in different tests. To be able to assess the impact of clustering, the table also includes pure message logging, *i.e.* 512 clusters, and the case where all inter-node messages are logged, *i.e.* 64 clusters.

Table 1 shows a well-known result [30]: a hybrid approach provides failure containment while dramatically reducing the amount of logged data compared to the basic message logging protocol. In the case of MiniGhost which is the most communication intensive application, using 16 clusters divides the maximum amount of data to log per process by 3($2.1MB/s$ vs $6.3MB/s$) while ensuring that only 6.25% (1/16) of the processes will have to restart should a failure occur. It can also be noticed that on our testbed, logging all inter-node messages will be in most cases two times as costly in terms of memory usage per process as having 16 clusters (which corresponds to 4 nodes per cluster).

In addition to these known facts, our experiments show that the growth rate of the logs is not uniform on all processes. Depending on the communication pattern of the application, some processes log much more data than others. Assuming that each process is provided with the same amount of memory, and that the application memory footprint is very similar for all processes, the maximum growth rate is probably the most important value to consider when choosing a clustering configuration since this process will run out of memory first. Note that for all applications and all numbers of clusters, the growth rate for the processes that log the least amount of data is always bellow 0.1 MB/s (except for MILC with more than 8 clusters).

Next, the experiments show that although the average amount of logged data generally grows with the increase in the number of clusters, it is not the case for the maximum logs growth rate. For instance, in the case of GTC, the maximum growth rate with 16 clusters is not bigger than with 2 clusters. Therefore, the configuration with 16 clusters seems to be better than the one with only 2 clusters. This might be partially due to the fact that the clustering technique we used aims at minimizing the total amount of data logged [30]. Other techniques that minimize the maximum growth rate of the logs should be studied. However it should still be noticed that even in the case of basic message logging or when all inter-node messages are logged, logging can be unbalanced between processes.

6.3 Failure-Free Performance

We evaluate the overhead introduced by SPBC in failure-free execution by comparing its performance to the performance of the native MPICH-3.0.2 library. We run experiments with 2, 4, 8, and 16 clusters. Table 2 presents the overhead observed with 16 clusters. It is the configuration that logs most messages, but the overhead is at most 1%. For lower number of clusters, we observed even smaller overhead. As it was already shown with other protocols employing message logging [7, 19, 29], the cost of logging message payload in the sender's memory has almost no impact on failure-free execution. We refer the reader to [19] for an evaluation of our logging technique over a network with higher performance (Myrinet MX-10G): The observed performance is similar to the one presented in Table 2.

AMG	CM1	GTC	MILC	MiniFE	MiniGhost
0.26%	0.63%	1.14%	0.07%	0.08%	0.36%

Table 2: failure-free overhead of SPBC in percent (16 clusters)

6.4 Recovery Performance

The goal of the experiments we present is to evaluate the time needed to re-execute the computation lost due to a failure, *i.e.* the *rework* time, and to compare it with the time needed to do the same computation during a failure-free execution. Due to current limitation of our prototype (no support for partial restart), we cannot simulate failures. Instead, we evaluate our protocol in recovery in the following way. We first execute the application with the chosen clustering configuration once to generate the logs and we save them to the local storage of the nodes. Then we restart the application and *simulate* the recovery of one cluster (the cluster including rank 0). It means that only the processes of this cluster are really executed. Other processes simply read the log files at the beginning of the execution, compute the lists of logged messages to be replayed and then start replaying them using the data available on the disk. If partial restart were already supported by SPBC, the processes of the non-failed clusters would not restart and would have all logged messages required for the recovery available in their memory. Note that if non-failed processes could continue progressing without waiting for the rolled-back ones to recover, it could impact recovery performance since these processes would not be always available to replay logged messages. However, it is not the case with the applications we consider since the processes are tightly coupled: Non-failed processes will need messages from the recovering processes

to progress. That is why we decided not to study this case. Note also that we do not evaluate the time required to restore the rolled back processes' state from the last checkpoint - this time is the same for any hybrid protocol.

Figure 5 shows the performance results in recovery with 2, 4, 8 and 16 clusters. In all cases, the rework time is lower than the failure-free execution time with MPICH. Processes can execute faster in recovery for two reasons. First they can skip the sending of inter-cluster messages. Second, when small messages are replayed from logs, they may be sent in advance during recovery, and so, be available to the receiver immediately when it tries to receive them. These two reasons explain why for configurations with smaller clusters where the amount of inter-cluster communication is bigger, the performance in recovery is better. We used the IPM profiling tool[4] to investigate the results in more details.

First, it should be mentioned that the speedup observed in recovery heavily depends on the computation to communication ratio of the application, since performance improvement can only be gained on the communication part. Three of the applications (CM1, GTC, and MiniFE) spend less that 10% of their time on communication and consequently recovery was at best only 4% faster than the failure-free execution. On the contrary, AMG spends more than 50% of its time on communication and achieves a maximum speedup of 25% in recovery. However, there are other factors that impact recovery performance. Even when the communication ratio is high, if the most time is spent in intra-cluster communication, then only a small speedup can be expected. This is what we observed in MILC and MiniGhost. AMG, on the contrary, was spending a lot of time in inter-cluster communication during the failure-free execution. Finally, it should also be mentioned that the performance in recovery is limited by the slowest process to recover. For instance, in the case of CM1, one process in the cluster does not communicate with processes from other clusters. It means that this process does not execute faster during recovery, and so, it limits the recovery performance.

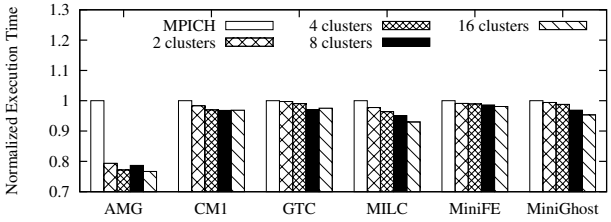

Figure 5: Performance of SPBC in Recovery

6.5 Distributed vs Centralized Recovery

We compared SPBC with HydEE [19]. To our knowledge, this is the only other existing protocol providing failure containment without logging any information reliably during the failure-free execution. HydEE requires the use of an additional process (the coordinator) to orchestrate the recovery and avoid mismatches: it notifies a process that it can replay the next message from logs once the recovering processes have acknowledged that all the inter-cluster messages that this message depends on have been replayed.

We could only use some of the NAS benchmarks (BT, LU, MG, SP) for this experiment because of the limitations of the HydEE prototype. Results are presented in Figure 6. All tests are run with 512 MPI ranks partitioned into 8 clusters. The Figure shows recovery time normalized to failure-free execution with MPICH. As we see, SPBC noticeably (up to factor of 2) outperforms HydEE, which in some cases runs even slower than the failure-free execution. The reason for this slowdown is the impact of the centralized coordination between processes during recovery.

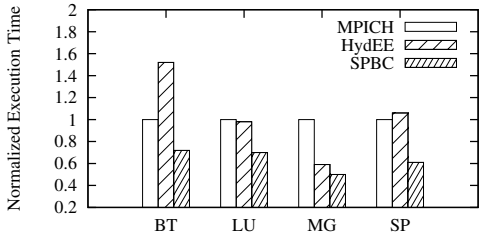

Figure 6: Comparison of the performance of HydEE and SPBC in recovery (8 clusters)

6.6 Discussion

The results presented in this section demonstrate the efficiency of SPBC both in failure-free execution and in recovery. Despite the limited scale of our testbed, we think that the presented experiments show the benefits of our approach. The performance of HydEE at larger scale could only get worse because of the centralized coordination algorithm it requires for recovery. On the other hand, the fully distributed design of SPBC can be observed in Algorithm 1: The whole algorithm (failure-free and recovery) is applied independently on each communication channel. This will allow SPBC to scale to a high number of nodes.

The experiments with SPBC also show that the use of a hierarchical checkpointing protocol involves some tradeoffs. Smaller clusters may allow recovering faster because of larger number of inter-cluster messages. But it also means that more data is logged, and so, that the protocol has a larger memory footprint. It would be interesting to study to which extent logging more messages can be beneficial for recovery. Note also that the clustering strategy we used in this paper is probably not the best one for recovery performance since it aims at minimizing the total amount of data logged, resulting in very imbalanced configurations with respect to logging: Inside one cluster some processes have a lot of communication with other clusters while others do not have any. A clustering strategy that would provide more balanced configurations might be beneficial for recovery performance. Studying all these questions is part of our future work.

7. THE CASE OF HYBRID PROGRAMMING MODELS

The results presented in Section 6 make SPBC an appealing solution for fault tolerance at large scale. However, SPBC assumes an MPI-only model. The current trend

[4]http://ipm-hpc.sourceforge.net/

in supercomputer architectures is increasing the number of cores per node. As a consequence, the MPI-only model used by many production scientific applications is being replaced by a hybrid model where one MPI process per node is used and intra-node parallelism is achieved through multi-threading [12]. The question whether the property our protocol relies on, namely channel-determinism, will remain valid for hybrid applications is open but we will now try to answer it partially. For now we cannot give a complete answer because we could not get access to a sufficient number of applications that use MPI + Threads. We also note that the discussion on the support for multi-threading in the context of MPI-3 is still open [1].

We discuss the two main levels of support for threads described in the latest MPI specification: `MPI_THREAD_FUNNELED` and `MPI_THREAD_MULTIPLE`. In `MPI_THREAD_FUNNELED`, a single thread is allowed to execute MPI calls. Legacy applications that combine MPI and OpenMP often use this mode. In this case, only computations are parallelized over multiple OpenMP threads, and so, it does not impact the communication behavior of the application[5]. Thus, SPBC will work for such applications as is.

In the case of `MPI_THREAD_MULTIPLE`, all threads may execute MPI calls. Since we only got access to few such applications we cannot draw general conclusions. However, we noticed that in many cases programmers try to use some mechanisms, namely communicators or tags, to differentiate between messages sent by different threads of the same MPI process. If communicators are used, our protocol could be used as is, of course assuming that threads have a channel-deterministic behavior, since we defined a channel in the context of a communicator. If tags are used and multiple threads can send over the same channel then channel-determinism will be lost: We cannot assume a total order of the messages sent on a channel anymore. Assuming that the application keeps some form of determinism, the problem is then finding out which messages need to be resent on which channel if a recovery takes place. One possible solution is to associate a sequence number with each (channel,tag) tuple instead of a single sequence number per channel as in the current version of SPBC. An alternative would be to generalize the use of pattern descriptions in the code so that after a failure, each process could precisely say which iteration of which pattern it is currently executing. Such solutions would need to be studied further.

8. CONCLUSION

Focusing on the properties of MPI HPC applications, we presented a checkpointing protocol that combines an unprecedented set of features. Our evaluations run with a representative set of HPC workloads show that (i) it provides almost no overhead in failure-free execution (considering only the overhead of message logging), (ii) it allows recovering efficiently after a failure, and (iii) it provides perfect failure containment. The combination of these three properties makes SPBC a good candidate for fault tolerance at extreme scale. It may require some modifications of the applications, but our experience shows that these changes are usually small and easy to apply.

[5]Except if introducing multi-threading changes the result of some computation (round-off errors, etc.).

To achieve these results, we identified a property common to many HPC applications, namely *channel-determinism*, and designed a protocol that makes use of this property. Instead of a solution that would work for all message passing applications, we find a solution that can be very efficient for many HPC use-cases. Additionaly, we introduced the *always-happens-before* relation as a new partial order relation on the events of a *channel-deterministic* application. The *always-happens-before* relation expresses the fact that MPI HPC applications are mostly composed of a set of well-defined computation steps that may include several iterations, and that, in each of these iterations, a well-defined set of communications is involved. The channel-determinism acknowledges that the execution of a process in most scientific applications does not depend on the relative order of messages received from different sources.

We think that taking the properties of target applications into account is the way to design fault tolerance solutions that can cope with the scale and failure rate of future exascale systems. In some sense, this approach is similar to the one adopted by application-based fault tolerance [13]. This paper illustrates how such an approach can be applied to checkpointing protocols.

9. ACKNOWLEDGEMENTS

The authors would like to thank Darko Petrović for his useful comments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This work was supported by INRIA-Illinois-ANL Joint Laboratory for Petascale Computing, the ANR RESCUE project, and the G8 ECS (Toward Exascale Climate Simulations) project.

10. REFERENCES

[1] Mpi3 hybrid working-group.

[2] A. H. Baker, R. D. Falgout, and U. M. Yang. An assumed partition algorithm for determining processor inter-communication. *Parallel Computing*, 32(5-6):394–414, June 2006.

[3] L. Bautista-Gomez, N. Maruyama, D. Komatitsch, S. Tsuboi, F. Cappello, S. Matsuoka, and T. Nakamura. FTI: high performance Fault Tolerance Interface for hybrid systems. In *IEEE/ACM SuperComputing 2011*, Seatle, USA, November 2011.

[4] L. Bautista-Gomez, T. Ropars, N. Maruyama, F. Cappello, and S. Matsuoka. Hierarchical Clustering Strategies for Fault Tolerance in Large Scale HPC Systems. In *IEEE Cluster 2012*, 2012.

[5] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the Message Logging Model for High Performance. *Concurrency and Computation : Practice and Experience*, 22:2196–2211, 2010.

[6] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello. Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. In *Proceedings of the 19th IEEE InternationalParallel and Distributed Processing Symposium (IPDPS'05)*, volume 1, page 97, April 2005.

[7] A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Correlated Set Coordination in Fault Tolerant Message Logging Protocols. In *Proceedings of the 17th international conference on Parallel processing*, Euro-Par'11, pages 51–64, 2011.

[8] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, USA, 2009.

[9] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *19th International Conference on Computer Communications and Networks (ICCCN 2010)*, 2010.

[10] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *IEEE/ACM SuperComputing 2012*, SC '12, pages 58:1–58:11, 2012.

[11] J. Dongarra, P. Beckman, T. Moore, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25:3–60, 2011.

[12] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 11–20, 2010.

[13] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 225–234, 2012.

[14] E. N. Elnozahy et al. System Resilience at Extreme Scale. Technical report, DARPA, 2008.

[15] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[16] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *IEEE/ACM SuperComputing 2011*, pages 44:1–44:12, 2011.

[17] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *IEEE/ACM SuperComputing 2012*, pages 78:1–78:12, 2012.

[18] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications. In *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS2011)*, Anchorage, USA, 2011.

[19] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications. In *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS2012)*, Shanghai, China, 2012.

[20] V. E. Henson and U. M. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, Apr. 2002.

[21] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[22] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1150–1158, 1986.

[23] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[24] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[25] E. Meneses, C. L. Mendes, and L. V. Kale. Team-based Message Logging: Preliminary Results. In *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010)*, May 2010.

[26] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. www.mpi-forum.org, 1995.

[27] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[28] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, 2007.

[29] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges. Alleviating scalability issues of checkpointing protocols. In *IEEE/ACM SuperComputing 2012*, SC '12, pages 18:1–18:11, 2012.

[30] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello. On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications. In *Proceedings of the 17th international conference on Parallel processing*, Euro-Par'11, pages 567–578, 2011.

[31] T. Ropars and C. Morin. Active optimistic and distributed message logging for message-passing applications. *Concurrency and Computation: Practice and Experience*, 23(17):2167–2178, 2011.